Design of a Sliding Window over Distributed and Asynchronous Event Streams

Yiling Yang, *Student Member, IEEE*, Yu Huang, *Member, IEEE*, Jiannong Cao, *Senior Member, IEEE*, Xiaoxing Ma, *Member, IEEE*, and Jian Lu

Abstract—The event stream model of computation has a wide range of applications, e.g., computer system monitoring, physical environment sensing/surveillance, and stock trade monitoring. Sliding windows are widely used to facilitate effective event stream processing. However, it is greatly challenged when the event sources are distributed and asynchronous. One important technique to cope with the asynchrony is to utilize that the meaningful snapshots of an asynchronous computation form a distributive lattice. It thus becomes the central challenge whether this lattice structure still preserves and how to maintain it at runtime, when we restrict our attention to events within sliding windows. To address this challenge, we first prove that the snapshots of the asynchronous event streams within the sliding windows form a convex distributive lattice (denoted by *Lat-Win*). This enables us to easily integrate existing predicate specification and detection techniques, to express and monitor properties of our concern over asynchronous event streams. Then we propose an algorithm to maintain *Lat-Win* at runtime. The proposed scheme is evaluated in a context-aware smart office scenario, where activities of the user can be recognized by monitoring multiple streams of sensed events. The *Lat-Win* algorithm is implemented on the open-source context-aware middleware we developed. The evaluation results first show the advantage of adopting sliding windows over asynchronous event streams. Then they show the performance of detecting specified predicates within *Lat-Win*, with dynamic changes in the computing environment.

Index Terms—Sliding window, lattice of snapshots, predicate detection, asynchronous event streams

1 Introduction

Event stream monitoring has a large class of both well-established and emerging applications, including: monitoring a large computing (nowadays often cyber-physical [1]) system, for malfunctioning diagnosis, performance analysis and tuning, etc. [2], [3], [4], [5]; physical environment sensing and surveillance in mobile pervasive computing scenarios [6], [7], [8]; real-time monitoring of stock trading and electric commerce [9].

Event streams are often generated from multiple distributed sources [10], [11]. More importantly, the event sources may not have global clocks or shared memory [6], [7], [12], [13]. The heterogeneity and constrained resources of event sources may lead to diverse and unpredictable computation delay. The growing adoption of mobile devices and wireless communications may result in unpredictable communication delay [12], [6], [2], [14], [15].

The asynchrony among the event sources makes it challenging to reason about properties of users' concerns over such *asynchronous event streams* [11], [6], [7], [8]. For example, we need to design and implement a distributed algorithm over a group of mobile robots, based on which the robots can coordinate to perform some task. It is notoriously difficult to design, implement, and debug such distributed programs [2]. It is because the robots work in a fully-distributed way, and in realistic settings, the robots may fail and only have clocks which are not perfectly synchronized [2]. One important technique to facilitate building such systems is to collect, replay, and query about properties of concern over the traces, i.e., the asynchronous event streams generated by the mobile robots [6], [7].

In another smart office scenario, a context-aware middleware may receive the event stream of user's location updates from his mobile phone (we assume that the user's location can be decided by the access point his phone connects to) [6]. The middleware may also receive event streams from sensors in the meeting room about whether there is a presentation going on. Sensors and the phone do not necessarily have synchronized clocks. Due to the asynchrony among them, the middleware cannot easily decide the composite global event ''*the user is in the meeting room, where a presentation is going on*'', to mute the phone intelligently.

The issue of how to cope with the asynchrony has been widely studied in distributed computing [16], [17]. One important approach relies on the ''happen-before'' relation resulting from message passing [18]. Based on this relation, various types of logical clocks can be devised [17], [19]. With logical time, one key notion in an asynchronous system is that all the global snapshots of the system form a distributive lattice [16], [17]. The lattice serves as the basis

- *Y. Yang, Y. Huang, X. Ma, and J. Lu are with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu Province 210046, China. E-mail: csylyang@gmail.com; {yuhuang, xxm, lj}@nju.edu.cn.*
- *J. Cao is with the Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong, China. E-mail: csjcao@comp.polyu.edu.hk.*

Manuscript received 12 Mar. 2013; revised 27 July 2013; accepted 5 Sept. 2013. Date of publication 15 Sept. 2013; date of current version 17 Sept. 2014. Recommended for acceptance by E. Leonardi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.233

TABLE 1
Notations Used in the Design of Lat-Win

Notation	Explanation
n	number of non-checker processes
$P^{(k)}, P_{che}$	non-checker / checker process ($1 \leq k \leq n$)
$e_i^{(k)}, s_i^{(k)}$	event / local state on $P^{(k)}$
$Que^{(k)}$	queue of local states from each $P^{(k)}$ on P_{che}
$W^{(k)}$	sliding window on a single event stream
$W_{min}^{(k)}/W_{max}^{(k)}$	the oldest/latest local state within $W^{(k)}$
w	uniform size of every $W^{(k)}$
W	n-dimensional sliding window over asynchronous event streams
$\mathcal{G}$	global state of asynchronous event streams
$\mathcal{C}$	Consistent Global State (CGS)
$\mathcal{C}[k]$	k^{th} constituent local state of CGS $\mathcal{C}$
LAT	original lattice of CGSs when no sliding window is used and the entire asynchronous event streams are processed
Lat-Win	lattice of CGSs within the n-dimensional sliding window
$\mathcal{C}_{min}, \mathcal{C}_{max}$	the minimal/maximal CGS in *Lat-Win*

for the specification of logic predicates, which delineate the application's concerns about properties of the asynchronous event streams [16], [17]. Manipulation of the lattice at runtime is often necessary to detect user-specified properties [20], [7], [8].

During the monitoring, the events may quickly accumulate to a huge volume, and so will the lattice of snapshots of the asynchronous event streams [17], [21]. Processing the entire event streams is often infeasible and, more importantly, not necessary [22]. In such applications, we often only need the most recent events. This can be effectively captured by the notion of a *sliding window* [23], [11], [22]. Processing events within the window can greatly reduce the processing cost. For example in the scenario of programming mobile robots, only the last ten updates of network connections of the robots may affect the current task, while in the smart office scenario, only the last five location updates of the user may be useful. Thus, we can restrict our attentions within the sliding windows over the asynchronous event streams.

Challenge of the asynchrony and effectiveness of sliding windows motivate us to study how to enable detection of properties over asynchronous event streams within sliding windows. Specifically, in a system of n asynchronous event streams and one sliding window on each stream, we define an *n-dimensional sliding window* as the Cartesian product of the window on every event stream. Considering the system of asynchronous event streams within the n-dimensional sliding window, we want to investigate whether the lattice structure of snapshots preserves. If it does, we need to further investigate how to effectively maintain this lattice of snapshots at runtime, to support effective detection of predicates over event streams within the window. Toward these problems, the contribution of this work is two-fold:

- We first prove that the lattice structure does preserve as for the global snapshots of asynchronous event streams within the n-dimensional sliding window. This enables us to easily integrate existing predicate specification and detection schemes, e.g., [7], [8], [20], [21]. More specifically, we show that the snapshots of the asynchronous event streams within the sliding windows form a convex distributive sublattice (denoted by *Lat-Win*) of the ''original lattice'' of the entire event streams;
- Then we characterize how *Lat-Win* evolves when the n-dimensional sliding window slides over the asynchronous event streams, based on which we propose an algorithm to maintain *Lat-Win* at runtime. We also show how *Lat-Win* can be easily integrated with existing predicate detection schemes.

A case study of a smart office scenario is conducted to demonstrate how our proposed *Lat-Win* facilitates context-awareness in asynchronous pervasive computing scenarios [6]. The *Lat-Win* maintenance algorithm is implemented and evaluated over MIPA—the open-source context-aware middleware we developed [24], [6], [7]. The performance measurements first show the advantage of adopting sliding windows over asynchronous event streams. Then the measurements show that using sliding windows, fairly accurate predicate detection (accuracy up to 95 percent) can be achieved, while the cost of event processing can be greatly reduced (to less than 1 percent).

The rest of this paper is organized as follows. Section 2 presents the preliminaries. Section 3 overviews how *Lat-Win* works, while Sections 4 and 5 detail the theoretical characterizations and algorithm design, respectively. Section 6 presents the experimental evaluation. Section 7 reviews the related work. Finally, Section 8 concludes the paper and discusses the future work.

2 Preliminaries

In this section, we first describe the system model of distributed and asynchronous event streams. Then we discuss the lattice of snapshots of asynchronous event streams. Finally, we introduce the *n-dimensional sliding window* over asynchronous event streams. Notations used in this work are listed in Table 1.

2.1 A System of Distributed and Asynchronous Event Streams

When monitoring a distributed system, we are faced with multiple distributed event sources which generate event streams at runtime. The event sources do not necessarily have global clocks or shared memory. They are modeled as n *non-checker processes* $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$ with message passing between each other [25], [6], [7]. Each $P^{(k)}$ produces a stream of *events* connected by *local states*: ''$e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, e_2^{(k)}, \ldots$'', as shown in Fig. 1a. The event may be local, indicating status update of the entity being monitored, or global, e.g., communication via sending/receiving messages. Different abstraction/filtering approaches according to different requirements can be adopted to abstract/filter the events. The non-checker processes form a loosely-coupled message-passing system. We assume that no messages are lost, altered, or spuriously introduced, as in [26], [25]. The underlying communication channel is not necessarily FIFO. For example in a context-aware computing scenario [6], non-checker processes may be deployed on the mobile

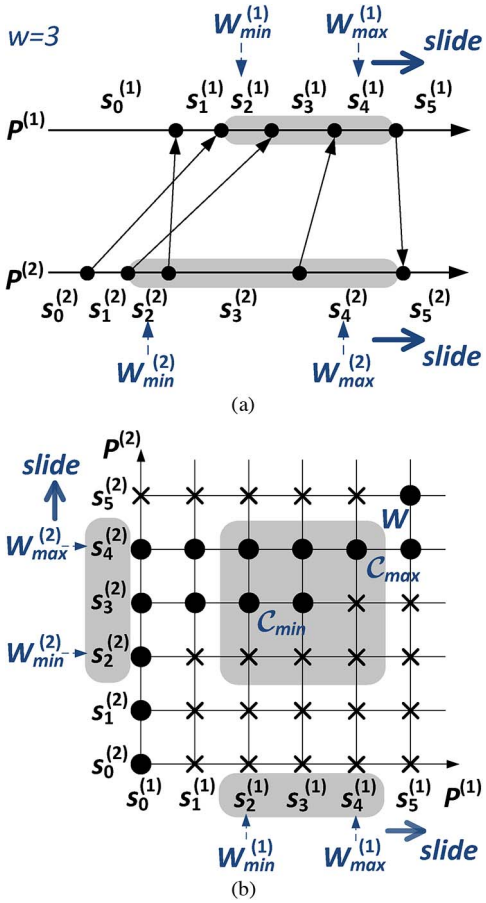


Fig. 1. System model. (a) Sliding windows over asynchronous event streams; (b) the n-dimensional sliding window over the lattice.

phones of the users. Status updates of the phones are modeled as events. Such events may be disseminated to the context-aware middleware to facilitate further context reasoning.

We re-interpret the notion of time based on Lamport's definition of the *happen-before* relation (denoted by '$\rightarrow$') resulting from message causality [18]. This happen-before relation can be effectively encoded and decoded based on the logical vector clock scheme [19], [6], [7]. Specifically, for two events e_1 and e_2, we have $e_1 \rightarrow e_2$ iff

- e_1 and e_2 are on the same non-checker process and e_1 is generated before e_2, or
- e_1 and e_2 are on different non-checker processes, and e_1 and e_2 are the corresponding sending and receiving of the same message, or
- there exists some e_3 such that $e_1 \rightarrow e_3 \rightarrow e_2$.

For two local states s_1 and s_2, $s_1 \rightarrow s_2$ iff the ending of s_1 happen-before (or coincides with) the beginning of s_2 (note that the beginning and ending of a state are both events). As shown in Fig. 1a, $s_2^{(2)} \rightarrow s_1^{(1)}$ and $s_4^{(1)} \rightarrow s_5^{(2)}$.

The non-checker processes send their local event streams to a *checker process* P_{che}, which is in charge of collecting and processing the asynchronous event streams [25], [6], [7], [8]. The checker process is a logical process, which can be deployed on one of the processes of the distributed system (i.e., on the same physical device with some non-checker process) or an external observer of the distributed system (i.e., on a separate and dedicated device). Also in the context-aware computing scenario [6], [7], P_{che} may be a context reasoning process deployed over the context-aware middleware. In a supply chain management scenario [27], P_{che} may be a central administration application, monitoring the progresses of multiple supply chains.

Whenever $P^{(k)}$ generates a new event and proceeds to a new local state, it sends the local state with the vector clock timestamp to P_{che}. We use message sequence numbers to ensure that P_{che} receives messages from each $P^{(k)}$ in FIFO manner [26], [25], [6], [7].

2.2 Lattice of Consistent Global States

In the tracking/monitoring application, we are concerned with the entity states after specific events are generated. For a system of asynchronous event streams, we are thus concerned with the global states or snapshots of the whole system. A global state $\mathcal{G} = (s^{(1)}, s^{(2)}, \ldots, s^{(n)})$ of asynchronous event streams is defined as a vector of local states from each $P^{(k)}$. A global state may be either consistent or inconsistent. The notion of *Consistent Global State (CGS)* is crucial in processing of asynchronous event streams. Intuitively, a global state is consistent if an omniscient external observer could possibly observe that the system enters this state. Formally, a global state $\mathcal{C}$ is *consistent* iff the constituent local states are pairwise concurrent [16], i.e.,

$$\mathcal{C} = \left(s^{(1)}, s^{(2)}, \ldots, s^{(n)}\right), \quad \forall i, j : i \neq j :: \neg\left(s^{(i)} \rightarrow s^{(j)}\right).$$

The CGS denotes a global snapshot or meaningful observation of the system of asynchronous event streams.

It is intuitive to define the *precede* relation (denoted by '$\prec$') between two CGSs: $\mathcal{C} \prec \mathcal{C}'$ if $\mathcal{C}'$ is obtained via advancing $\mathcal{C}$ by exactly one local state on one non-checker process. The *lead-to* relation (denoted by '$\rightsquigarrow$') is defined as the transitive closure of '$\prec$' [16], [17].

The set of all the CGSs together with the '$\rightsquigarrow$' relation form a distributive lattice [16], [17], [28]. As shown in Fig. 1b, black dots denote the CGSs and the edges between them depict the '$\prec$' relation. The crosses '$\times$' denote the inconsistent global states. The lattice structure serves as a key notion for the detection of global predicates over asynchronous event streams [16], [17].

2.3 The n-Dimensional Sliding Window over Asynchronous Event Streams

On P_{che}, local states of each event source $P^{(k)}$ are queued in $Que^{(k)}$. As discussed in Section 1, in many cases, it is too expensive and often unnecessary to process the entire event streams. A *local sliding window* $W^{(k)}$ of size w is imposed on each $Que^{(k)}$. Then we can define the *n-dimensional sliding window* W as the Cartesian product of each $W^{(k)}$:

$$W = W^{(1)} \times W^{(2)} \times \cdots \times W^{(n)}.$$

As shown in Fig. 1a, the window $W^{(1)}$ with $w = 3$ on $P^{(1)}$ currently contains $\{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}\}$. The 2-dimensional sliding window $W^{(1)} \times W^{(2)}$ is depicted by the gray square in Fig. 1b. The arrival of $s_5^{(1)}$ will trigger the 2-dimensional

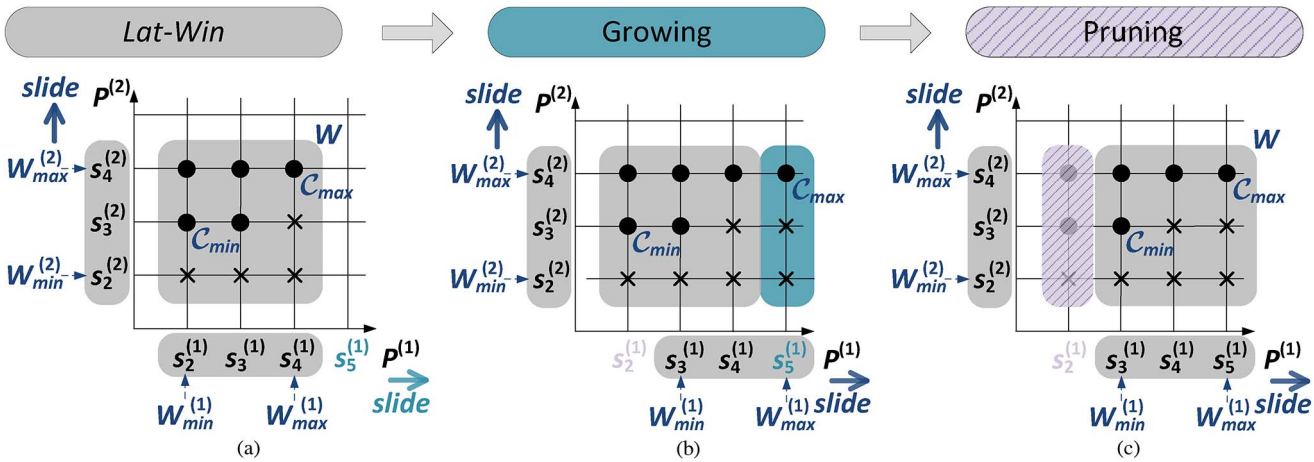


Fig. 2. Online maintenance of *Lat-Win*. When $s_5^{(1)}$ arrives, *Lat-Win* first grows with a set of new CGSs and then prunes the stale CGSs. (a) *Lat-Win* induced by $\{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}, s_2^{(2)}, s_3^{(2)}, s_4^{(2)}\}$, and $s_5^{(1)}$ arrives. (b) *Lat-Win* grows with the CGSs which contain $s_5^{(1)}$ as a constituent, and $\mathcal{C}_{\max}$ is updated. (c) *Lat-Win* prunes the CGSs which contain $s_2^{(1)}$ as a constituent, and $\mathcal{C}_{\min}$ is updated.

window to slide in the dimension of $P^{(1)}$, and $W^{(1)}$ will be updated to $\{s_3^{(1)}, s_4^{(1)}, s_5^{(1)}\}$.

We assume that the concurrency control scheme is available on P_{che}, which means that the events from all non-checker processes are processed one at a time. We also assume that the sliding windows have a uniform size w. Note that this assumption is not restrictive and is for the ease of interpretation. Our proposed scheme also works if the local windows on different streams have different sizes.

3 Lat-Win—Design Overview

The key notion in coping with the asynchrony is that the meaningful snapshots of an asynchronous computation form a lattice. Now we restrict our attention to the n-dimensional sliding window over a collection of n asynchronous streams. Accordingly, the central problem in this work is to prove that the lattice structure preserves within the n-dimensional sliding window, and to design an algorithm to maintain the lattice of snapshots within the window at runtime. Toward this problem, our contribution is two-fold, as overviewed below and detailed in Sections 4 and 5.

3.1 Characterization of Lat-Win

An important property concerning *Lat-Win* is that all the CGSs within the n-dimensional sliding window together with the '$\rightsquigarrow$' relation have the lattice structure. Moreover, *Lat-Win* turns out to be a distributive convex sublattice of the original lattice *LAT* (the lattice obtained when no sliding window is used and all events in all the streams are processed). As shown in Fig. 1b, the gray square in the middle is a 2-dimensional sliding window over two asynchronous event streams produced by $P^{(1)}$ and $P^{(2)}$. The CGSs within the square form a convex sublattice of the original lattice *LAT*, i.e., the *Lat-Win*.

When an event $e_j^{(i)}$ is generated on $P^{(i)}$ and the corresponding local state $s_j^{(i)}$ arrives at P_{che}, the stale local state $s_k^{(i)}$ $(j-k=w)$ in window $W^{(i)}$ will be discarded. The *Lat-Win* will ''grow'' with a set of CGSs consisting of $s_j^{(i)}$ and other local states from $W^{(m)}(m \neq i)$, and ''prune'' the CGSs which contain the stale local state $s_k^{(i)}$ as a constituent.

For example, assume that the *Lat-Win* is initially shown in Fig. 2a. When $s_5^{(1)}$ arrives, $s_2^{(1)}$ will be discarded, and $s_5^{(1)}$ will be combined with local states in $W^{(2)}$ to obtain the CGS $\mathcal{C}_{5,4} = (s_5^{(1)}, s_4^{(2)})$ in the blue rectangle in Fig. 2b. CGSs which contain $s_2^{(1)}$ as a constituent in the left shaded rectangle of Fig. 2c will be discarded. The CGSs in the current window (e.g., the gray square in Fig. 2c) remain to be a sublattice. It can be envisioned as the 2-dimensional window containing the *Lat-Win* slides over the asynchronous event streams produced by $P^{(1)}$ and $P^{(2)}$.

3.2 Online Maintenance of Lat-Win

Based on the theoretical characterizations above, we propose an algorithm for runtime maintenance of *Lat-Win*. Let $\mathcal{C}_{\min}$ ($\mathcal{C}_{\max}$) denote the CGS with no predecessors (successors) in *Lat-Win*. Both $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ serve as two ''anchors'' in updating *Lat-Win*. When a local state arrives, *Lat-Win* ''grows'' from $\mathcal{C}_{\max}$ and ''prunes'' from $\mathcal{C}_{\min}$, as shown in Fig. 2. After the growing and pruning, $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ are also updated for further maintenance of *Lat-Win*. Due to the symmetry of the lattice structure, the growing and pruning of *Lat-Win* are dual. So are the updates of $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$.

4 Lat-Win—Characterizing the Snapshots of Windowed Asynchronous Event Streams

The theoretical characterizations of *Lat-Win* consist of two parts. First we study the lattice of snapshots within the n-dimensional sliding window. Then we study how the *Lat-Win* evolves as the n-dimensional window slides.

4.1 Sub-Lattice within the Sliding Window

An n-dimensional sliding window consists of n local windows sliding on event streams produced by $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$, and induces n segments of local states $W^{(1)}, W^{(2)}, \ldots, W^{(n)}$. The happen-before relation between local states has been encoded in their logical clock time-stamps. Based on the local states as well as the happen-before relation among them, we can get a set of CGSs

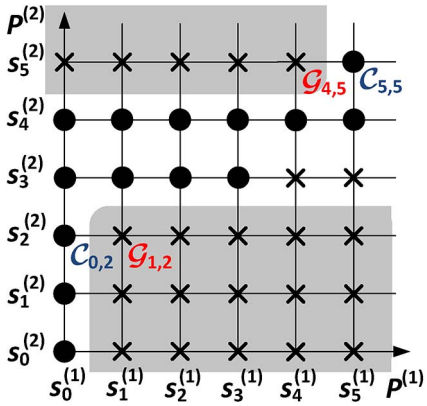


Fig. 3. The restrictions which have no CGSs induced by $\mathcal{G}_{1,2}$ and $\mathcal{G}_{4,5}$.

within the n-dimensional sliding window. An important property we find is that these CGSs together with the '$\rightsquigarrow$' relation also form a lattice—*Lat-Win*. This enables us to easily integrate existing predicate specification and detection schemes, e.g., [7], [8], [20], [21]. More importantly, *Lat-Win* is a *distributive convex sub-lattice* of the original lattice *LAT* [29]. Formally,

Theorem 1. *Given an n-dimensional sliding window $W = W^{(1)} \times W^{(2)} \times \cdots \times W^{(n)}$ over asynchronous event streams, let $Set_{\mathcal{C}}(W)$ denote the CGSs constructed from local states in W. If $Set_{\mathcal{C}}(W)$ is not empty,*

1. *$(Set_{\mathcal{C}}(W), \rightsquigarrow)$ forms a lattice, denoted by Lat-Win;*
2. *Lat-Win is a sublattice of LAT;*
3. *Lat-Win is convex and distributive.*[1]

The geometric interpretation of Theorem 1 is that W can be viewed as an n-dimensional ''cube'' over the original lattice *LAT*, and CGSs within the cube also form a lattice *Lat-Win*. Moreover, the ''convex'' and ''distributive'' properties of *LAT* preserve when we focus on CGSs within the cube. Let $\mathcal{C}_{i,j}$ denote the CGS $(s_i^{(1)}, s_j^{(2)})$. As shown in Fig. 2a, local windows $W^{(1)} = \{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}\}$ and $W^{(2)} = \{s_2^{(2)}, s_3^{(2)}, s_4^{(2)}\}$. They define a square on *LAT* (Fig. 1b) and induce a sublattice $Lat\text{-}Win = (\{\mathcal{C}_{2,3}, \mathcal{C}_{2,4}, \mathcal{C}_{3,3}, \mathcal{C}_{3,4}, \mathcal{C}_{4,4}\}, \rightsquigarrow)$. The induced sublattice is convex because all CGSs ''greater than'' $\mathcal{C}_{2,3}$ and ''smaller than'' $\mathcal{C}_{4,4}$ in the original lattice are contained in *Lat-Win*.

Given *Lat-Win* defined in Theorem 1, we further study how *Lat-Win* is contained in the cube. Is this cube a tight wrapper, i.e., does *Lat-Win* span to the boundary of the cube? First note that the maximal CGS and the minimal CGS are both important to the update of *Lat-Win*. Intuitively, the maximal CGS $\mathcal{C}_{\max}$ of *Lat-Win* is on the upper (right) bound $W_{\max}^{(i)}$ of at least one local window $W^{(i)}$, so that *Lat-Win* could grow with newly arrived local states from $P^{(i)}$. Dually, the minimal CGS $\mathcal{C}_{\min}$ is on the lower (left) bound $W_{\min}^{(j)}$ of at least one local window $W^{(j)}$, so that *Lat-Win* once grew from the stale local states from $P^{(j)}$ in the past. As shown in Fig. 2a, the maximal CGS $\mathcal{C}_{4,4}[1] = s_4^{(1)} = W_{\max}^{(1)}$, $\mathcal{C}_{4,4}[2] = s_4^{(2)} = W_{\max}^{(2)}$ and the minimal CGS $\mathcal{C}_{2,3}[1] = s_2^{(1)} = W_{\min}^{(1)}$. Formally,

Theorem 2. *If Lat-Win is not empty,*

1. $\exists\, i, \mathcal{C}_{\max}[i] = W_{\max}^{(i)}$;
2. $\exists\, j, \mathcal{C}_{\min}[j] = W_{\min}^{(j)}$.

4.2 Update of Lat-Win When the Window Slides

In this section, we discuss the update of *Lat-Win* when the n-dimensional window slides. Informally, the window slides as a new event is generated on $P^{(k)}$ and the corresponding local state arrives at P_{che}. When P_{che} receives a new local state from $P^{(k)}$, the stale local state (i.e., the old $W_{\min}^{(k)}$) will be discarded. *Lat-Win* will grow with the CGSs containing the newly arrived local state, and prune the CGSs containing the stale local state, as shown in Fig. 2. Since the intersection between the set of new CGSs and the set of stale CGSs is empty, the growing and pruning of *Lat-Win* can be processed in any order. In this work, we first add newly obtained CGSs to *Lat-Win* and then prune the stale CGSs. During the growing and pruning processes, $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ are also updated for further updates of *Lat-Win*. We characterize the evolution of *Lat-Win* in three steps:

- Lemma 3 defines the *restrictions* of lattice, which serve as the basis for further growing and pruning;
- Theorem 4 defines the condition when *Lat-Win* can grow and Theorem 5 identifies the new $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ when *Lat-Win* grows;
- Theorem 6 defines the condition when *Lat-Win* can prune and Theorem 7 identifies the new $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ when *Lat-Win* prunes.

The growing and pruning are dual, as well as the updates of $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$.

4.2.1 Restrictions

Before we discuss the update of *Lat-Win*, we first introduce the notion of a *restriction*. When we obtain a global state and decide that it is not consistent, we can induce a specific region containing only inconsistent global states. This specific region is called a *restriction* in [19].

The geometric interpretation can be illustrated by the example in Fig. 3. Global states $\mathcal{G}_{1,2} = (s_1^{(1)}, s_2^{(2)})$ and $\mathcal{G}_{4,5} = (s_4^{(1)}, s_5^{(2)})$ are not consistent ($s_2^{(2)} \rightarrow s_1^{(1)}$ and $s_4^{(1)} \rightarrow s_5^{(2)}$ in Fig. 1a). When looking from $\mathcal{C}_{0,2}$, $\mathcal{G}_{1,2}$ makes the lower gray region have no CGSs. When looking from $\mathcal{C}_{5,5}$, $\mathcal{G}_{4,5}$ makes the upper gray region have no CGSs. Formally,

Lemma 3. *Given a CGS $\mathcal{C}$ of a lattice, and two global states $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_1[i]$ ($\mathcal{G}_2[i]$) is the first (last) local state after (before) $\mathcal{C}[i]$ on $P^{(i)}$, $\forall\, k \neq i$, $\mathcal{G}_1[k] = \mathcal{G}_2[k] = \mathcal{C}[k]$,*

1. *If $\mathcal{G}_1$ is not a CGS, then $\exists\, j \neq i$, $\mathcal{G}_1[j] \rightarrow \mathcal{G}_1[i]$, and none of the global states in the following set is a CGS: $\{\mathcal{G} | (\mathcal{G}[j] \rightarrow \mathcal{G}_1[j] \vee \mathcal{G}[j] = \mathcal{G}_1[j]) \wedge (\mathcal{G}_1[i] \rightarrow \mathcal{G}[i] \vee \mathcal{G}_1[i] = \mathcal{G}[i])\}$;*
2. *If $\mathcal{G}_2$ is not a CGS, then $\exists\, j \neq i$, $\mathcal{G}_2[i] \rightarrow \mathcal{G}_2[j]$, and none of the global states in the following set is a CGS: $\{\mathcal{G} | (\mathcal{G}_2[j] \rightarrow \mathcal{G}[j] \vee \mathcal{G}_2[j] = \mathcal{G}[j]) \wedge (\mathcal{G}[i] \rightarrow \mathcal{G}_2[i] \vee \mathcal{G}[i] = \mathcal{G}_2[i])\}$.*

1. Please refer to the supplementary file which is available in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.233 for the proofs of all the theorems and lemmas.

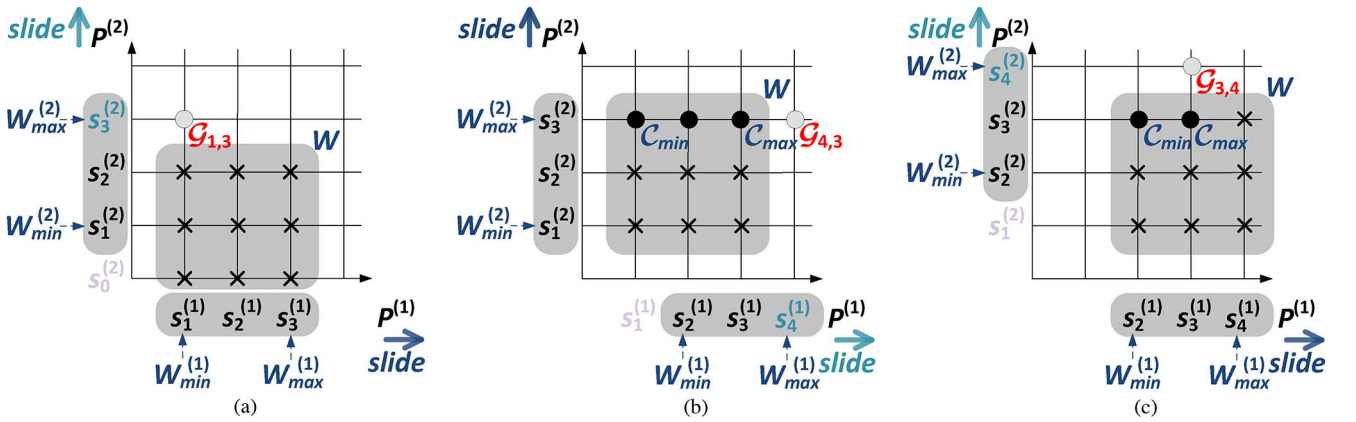


Fig. 4. The slide of the n-dimensional window. Assume the arrival of local states is $s_0^{(1)}, s_1^{(1)}, s_2^{(1)}, s_0^{(2)}, s_1^{(2)}, s_2^{(2)}, s_3^{(1)}, s_3^{(2)}, s_4^{(1)}, s_4^{(2)}, \ldots$. (a) The empty window induced by $\{s_1^{(1)}, s_2^{(1)}, s_3^{(1)}, s_0^{(2)}, s_1^{(2)}, s_2^{(2)}\}$, and $s_3^{(2)}$ arrives. (b) *Lat-Win* induced by $\{s_1^{(1)}, s_2^{(1)}, s_3^{(1)}, s_1^{(2)}, s_2^{(2)}, s_3^{(2)}\}$, and $s_4^{(1)}$ arrives. (c) *Lat-Win* induced by $\{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}, s_1^{(2)}, s_2^{(2)}, s_3^{(2)}\}$, and $s_4^{(2)}$ arrives.

4.2.2 Growing of Lat-Win

On the arrival of a new local state $s_i^{(k)}$, the n-dimensional window slides in the dimension of $P^{(k)}$, i.e., $W^{(k)}_{\max} = s_i^{(k)}$, and a set of newly obtained CGSs (containing $s_i^{(k)}$ as a constituent) will be added into *Lat-Win*. We find that the growing process does not have to explore the whole combinational space of the new local state with all local states from every other $W^{(j)}$. If *Lat-Win* is not empty, it will grow from $\mathcal{C}_{\max}$ in *Lat-Win*. The reason is that, if the next global state growing from $\mathcal{C}_{\max}$ is not consistent, as $\mathcal{G}_1$ in Lemma 3, it can be proved that the global states containing the newly arrived local state as a constituent are all in the restriction induced by a further global state and therefore not consistent. When *Lat-Win* is empty, the lattice can grow iff one CGS can be obtained containing the new local state and a lower bound $W^{(j)}_{\min}$ of some local window. This is because as discussed in Theorem 2, the new $\mathcal{C}_{\min}$ should contain at least a lower bound of a local window. Formally,

Theorem 4. *When a new event $e_i^{(k)}$ is generated on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives,*

1. *If Lat-Win $\neq \emptyset$, then Lat-Win can grow iff $\mathcal{C}_{\max}[k] = s_{i-1}^{(k)}$ (the old $W^{(k)}_{\max}$) and global state $\mathcal{G}(\mathcal{G}[k] = s_i^{(k)}, \forall j \neq k, \mathcal{G}[j] = \mathcal{C}_{\max}[j])$ is a CGS;*
2. *If Lat-Win $= \emptyset$, then Lat-Win can grow iff $\{\mathcal{C} | \mathcal{C}[k] = s_i^{(k)},\ \exists j \neq k,\ \mathcal{C}[j] = W^{(j)}_{\min},\ \mathcal{C}$ is a CGS$\} \neq \emptyset$.*

We illustrate the theorem by three examples in Fig. 4, on the arrival of $s_3^{(2)}$, $s_4^{(1)}$, and $s_4^{(2)}$. In Fig. 4a, the current *Lat-Win* is empty and $s_3^{(2)}$ arrives. The lattice can grow since $\mathcal{G}_{1,3}$ is a CGS. Thus *Lat-Win* grows to the new lattice in Fig. 4b. In Fig. 4b, the current *Lat-Win* is not empty and $s_4^{(1)}$ arrives. *Lat-Win* can grow iff $\mathcal{C}_{\max}[1] = s_3^{(1)}$ and $\mathcal{G}_{4,3}$ is a CGS. Note that $\mathcal{G}_{4,3}$ is not a CGS (in Fig. 3). Thus *Lat-Win* cannot grow, as shown in Fig. 4c. In Fig. 4c, the current *Lat-Win* is not empty and $s_4^{(2)}$ arrives. *Lat-Win* can grow iff $\mathcal{C}_{\max}[2] = s_3^{(2)}$ and $\mathcal{G}_{3,4}$ is a CGS. Note that $\mathcal{G}_{3,4}$ is a CGS (in Fig. 3). Thus *Lat-Win* can grow to the new lattice in Fig. 2a.

Both $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ are important to the update of *Lat-Win*. Thus, we discuss how to locate $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ after the growing of *Lat-Win* for further updates. After the growing of *Lat-Win*, the new $\mathcal{C}_{\max}$ should contain the new local state as its constituent. If *Lat-Win* was empty and grows with the new local state, $\mathcal{C}_{\min}$ should contain the new local state as its constituent. For example in Fig. 4a, *Lat-Win* is empty and can grow with the newly arrived local state $s_3^{(2)}$, the new $\mathcal{C}_{\max}[2] = s_3^{(2)}$ and the new $\mathcal{C}_{\min}[2] = s_3^{(2)}$, as shown in Fig. 4b. Formally,

Theorem 5. *When a new event $e_i^{(k)}$ is generated on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives,*

1. *If Lat-Win can grow, then $\mathcal{C}_{\max}[k] = s_i^{(k)}$ (the new $W^{(k)}_{\max}$); else $\mathcal{C}_{\max}$ remains not changed;*
2. *If Lat-Win $= \emptyset$ and can grow, then $\mathcal{C}_{\min}[k] = s_i^{(k)}$; else $\mathcal{C}_{\min}$ remains not changed.*

4.2.3 Pruning of Lat-Win

After the growing of new CGSs, *Lat-Win* will prune the CGSs which contain the stale local state. The pruning does not have to explore the whole lattice to check whether a CGS contains the stale local state. Intuitively, *Lat-Win* can prune, iff *Lat-Win* is not empty and $\mathcal{C}_{\min}$ contains the stale local state. Formally,

Theorem 6. *When a new event $e_i^{(k)}$ is generated on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives, after the growing, Lat-Win can prune, iff Lat-Win $\neq \emptyset$ and $\mathcal{C}_{\min}[k] \rightarrow W^{(k)}_{\min}$.*

For example, in Fig. 4b, on the arrival of $s_4^{(1)}$, $\mathcal{C}_{\min}[1] = s_1^{(1)}$ and $\mathcal{C}_{\min}[1] \rightarrow W^{(1)}_{\min}$. Thus, *Lat-Win* can prune, as shown in Fig. 4c.

We then discuss how to locate $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ after the pruning of *Lat-Win* for further updates. When a new local state from $P^{(k)}$ arrives, after the growing process, if *Lat-Win* prunes to be empty, $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ are *null*. If *Lat-Win* prunes to be not empty, $\mathcal{C}_{\min}$ should contain the new $W^{(k)}_{\min}$. Formally,

Theorem 7. *When a new event $e_i^{(k)}$ is generated on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives, after the growing,*

1. *If $\mathcal{C}_{\max}[k] \rightarrow W^{(k)}_{\min}$, then $\mathcal{C}_{\max} = null$; else $\mathcal{C}_{\max}$ remains not changed;*

2. *If* $\mathcal{C}_{\max}[k] \rightarrow W^{(k)}_{\min}$, *then* $\mathcal{C}_{\min} = null$; *if* $\mathcal{C}_{\max}[k] \not\rightarrow W^{(k)}_{\min}$ *and Lat-Win can prune, then* $\mathcal{C}_{\min}[k] = W^{(k)}_{\min}$; *else* $\mathcal{C}_{\min}$ *remains not changed.*

For example, in Fig. 4b, on the arrival of $s^{(1)}_4$, *Lat-Win* can prune and $\mathcal{C}_{\max}[1] \not\rightarrow W^{(1)}_{\min}$, then the new $\mathcal{C}_{\min}[1] = W^{(1)}_{\min}$ and $\mathcal{C}_{\max}$ is not changed, as shown in Fig. 4c.

5 Lat-Win—Online Maintenance Algorithm

In this section, we present the design of the *Lat-Win* maintenance algorithm, based on the theoretical characterizations above. Both $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ serve as two anchors in maintaining *Lat-Win*. When a new local state arrives, *Lat-Win* grows from $\mathcal{C}_{\max}$ and prunes from $\mathcal{C}_{\min}$. During the growing and pruning processes, $\mathcal{C}_{\min}$ and $\mathcal{C}_{\max}$ are also updated for further maintenance of *Lat-Win*.

P_{che} is in charge of collecting and processing the local states sent from non-checker processes. Upon initialization, P_{che} gets the window size w and initializes n local windows $W^{(k)}$. Upon receiving a new local state from $P^{(k)}$, P_{che} first enqueues the local state into $Que^{(k)}$ and then updates *Lat-Win* in the order of growing and pruning. After that, specific predicate detection algorithms (e.g., [7], [8], [20], [21]) can be easily integrated to detect predicates over *Lat-Win*. Pseudo codes of the maintenance algorithm are listed in Algorithm 1.

Algorithm 1: *Lat-Win* maintenance algorithm

```
1  Upon Initialization
2  get window size w, and initialize window buffers W^(k);
3  Upon Receiving local state (s_i^(k)) from P^(k)
4  Que^(k).enque(s_i^(k));
5  if s_i^(k) = Que^(k).head() then
6      pop the front continuous local states of
       Que^(k) to the end of InputQue;
7      trigger update();

   subroutine update()
1  while InputQue ≠ ∅ do
2      pop s_i^(k) = InputQue.head();
3      push s_i^(k) into W^(k) ;        /* W^(k)_max = s_i^(k) */
4      grow_lattice(s_i^(k), k) ;       /* Algorithm 2 */
5      prune_lattice(C_min, k) ;        /* Algorithm 3 */
6      trigger the pre-specified predicate detection
       algorithm;
```

5.1 Growing of Lat-Win

On the arrival of a new local state, the algorithm of growing consists of three steps. First, it is checked whether *Lat-Win* can grow, as discussed in Theorem 4. If yes, *Lat-Win* will grow with a set of new CGSs containing the newly arrived local state. During the step of growing, $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ are updated too, as discussed in Theorem 5. Pseudo codes of growing are listed in Algorithm 2.

Specifically, when *Lat-Win* is empty, the lattice can grow iff the set S in line 2 is not empty. If S is not empty, the $grow(\mathcal{C})$ sub-routine will be triggered to add the new CGSs. When *Lat-Win* is not empty, the lattice can grow iff $\mathcal{C}_{\max}$ and the next global state satisfy the condition defined in Theorem 4, as shown in line 4-7. If the condition is satisfied, the $grow(\mathcal{C})$ sub-routine will be triggered to add the new CGSs. The growing of *Lat-Win* is achieved by recursively adding all the predecessors and successors of a CGS, as shown in line 5-10 of the sub-routine. During the growing process, $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ are also updated in line 3-4 of the sub-routine. Theorem 5 ensures that $\mathcal{C}_{\max}$ can be found in the newly added CGSs, and that when *Lat-Win* was empty, $\mathcal{C}_{\min}$ can be found in the newly added CGSs.

Algorithm 2: $grow\_lattice(s_i^{(k)}, k)$

```
1  if Lat-Win = ∅ then
2      if S = {C | C[k] = s_i^(k), ∃j ≠ k, C[j] = W^(j)_min, C is
       a CGS} ≠ ∅ then
3          get a CGS C from S; grow(C);
4  else if C_max[k] = s_{i-1}^(k) then
5      combine C_max and s_i^(k) to get a global state G;
6      if G is a CGS then
7          connect G to C_max; grow(G);

   subroutine grow(C)
1  Set prec(C) = {C' | ∀i, C'[i] ∈ W^(i), C' is a CGS,
   C' ≺ C};
2  Set sub(C) = {C' | ∀i, C'[i] ∈ W^(i), C' is a CGS,
   C ≺ C'};
3  if prec(C) = ∅ then C_min = C;
4  if sub(C) = ∅ then C_max = C;
5  foreach C' in prec(C) do
6      if C' ∉ Lat-Win then
7          connect C' to C; grow(C');
8  foreach C' in sub(C) do
9      if C' ∉ Lat-Win then
10         connect C to C'; grow(C');
```

5.2 Pruning of Lat-Win

Dually, the algorithm of pruning consists of three steps as well. First, it is checked whether *Lat-Win* can prune, as discussed in Theorem 6. If yes, *Lat-Win* will prune the CGSs which contain the stale local state. During the step of pruning, $\mathcal{C}_{\max}$ and $\mathcal{C}_{\min}$ are updated too, as discussed in Theorem 7. Pseudo codes of pruning are listed in Algorithm 3.

Specifically, the lattice can prune iff the condition in line 1 is satisfied. If $\mathcal{C}_{\max}[k]$ is the stale local state, *Lat-Win* will prune to be empty, as shown in line 2-3. Otherwise, the CGSs which contain $\mathcal{C}[k]$ will be deleted, as shown in line 5-16. During the pruning process, $\mathcal{C}_{\min}$ is also updated in line 13-15. Theorem 7 ensures that $\mathcal{C}_{\min}$ can be found in the CGSs which contain the new $W^{(k)}_{\min}$.

5.3 Complexity Analysis

Assume that the space for storing a single CGS is one unit. The worst-case space cost of the original lattice LAT is

Algorithm 3: $prune\_lattice(\mathcal{C}, k)$

```
1  if Lat-Win ≠ ∅ ∧ C_min[k] → W_min^(k) then
2      if C_max[k] = C[k] then
3          Lat-Win = ∅, C_max = C_min = null;
4      else
5          Set S = {C};
6          while S ≠ ∅ do
7              pop C' from S;
8              Set sub(C') = {C''|C' ≺ C'', C'' is a CGS};
9              foreach C'' in sub(C') do
10                 delete the connection between C'
                   and C'';
11                 if C''[k] = C'[k] ∧ C'' ∉ S then
12                     add C'' into S;
13                 else if C''[k] = W_min^(k) then
14                     Set prec(C'') = {C'''|C''' ≺ C'', C'''
                       is a CGS} ;   /* without C' */
15                     if prec(C'') = ∅ then C_min = C'';
16             delete C';
```

$O(p^n)$, where n is the number of non-checker processes, and each non-checker process has $O(p)$ events. However, the worst-case space cost of *Lat-Win* is bounded by the window size w, that is, $O(w^n)$, where w is a predefined parameter and much less than p. Due to the incremental nature of Algorithm 2, the space cost of the incremental part of *Lat-Win* in each time of growing is $O(w^{n-1})$.

The worst-case time cost of growing (Algorithm 2) happens when all the global states in the blue rectangle in Fig. 2b are CGSs. Thus, the worst-case time of growing is $O(n^3 w^{n-1})$, where w^{n-1} is the number of the global states in the blue rectangle and n^3 is the time cost of checking whether a global state is consistent.

The worst-case time cost of pruning (Algorithm 3) happens when all the CGSs in the left shaded rectangle in Fig. 2c should be discarded. Thus, the worst-case time of pruning is $O(w^{n-1})$, where w^{n-1} is the worst-case number of the CGSs in the left shaded rectangle.

From the performance analysis we can see that, by tuning size of the sliding window, the cost of asynchronous event processing can be bounded. This justifies the adoption of sliding windows when only the recent part of the event streams are needed by the tracking/monitoring application, and the application needs to bound the cost of event processing.

6 Experimental Evaluation

A case study of a smart office scenario is conducted to demonstrate how our proposed *Lat-Win* facilitates context-awareness in asynchronous pervasive computing scenarios [6]. The *Lat-Win* maintenance algorithm is implemented and evaluated over MIPA—the open-source context-aware middleware we developed [24], [6], [7]. Please refer to Section 9 of the supplementary file available online for detailed discussions on the case study, experiment configurations, and evaluation results.

The performance measurements first show the benefits of adopting sliding windows over asynchronous event streams. By adopting a pretty small window size (4 in our case study), fairly accurate predicate detection (accuracy up to 95 percent) can be achieved, while the cost of event processing can be greatly reduced (to less than 1 percent). We further investigate the performance of *Lat-Win* by tuning the message delay, the window size, and the number of non-checker processes.

7 Related Work

The lattice of global snapshots is a key notion in modeling the behavior of asynchronous systems [17], [20], [30], [6], [7], [8], [31], and is widely used in areas such as distributed program debugging [26], [25] and fault-tolerant computing [32]. One critical challenge is that the lattice of snapshots evolves to exponential size in the worst-case. Various schemes are used to cope with the lattice explosion problem [26], [25], [31], [21], [33], [6], [7]. For example, in [31], the authors proposed the notion of ''computation slice'' to efficiently compute all global states which satisfy a regular predicate [31]. In [21], the authors discussed that certain (useless) part of the lattice can be removed at runtime to reduce the size of the lattice. Our previous work [7] is intended for the runtime detection of dynamic predicates (in the form of regular expressions) over the entire event streams from distributed event sources. In [7], we propose the notion of ''active surface'' of the lattice, which enables the incremental detection of dynamic predicates at runtime. In this work, we make use of the observation that, in many tracking/monitoring applications, it is often prohibitive and, more importantly, unnecessary to process the entire streams. Thus, we use sliding windows over distributed event streams to reduce the size of the lattice. We focus on the maintenance of *Lat-Win*, which is orthogonal to the detection of specific types of predicates.

The challenge of distributed and asynchronous event sources has recently been studied in event stream processing [34], [35], [10], [22], [23], [11]. Existing schemes for event stream processing implicitly assume that the distributed event sources have synchronized clocks, but the propagation of events suffers from uncertain delay [11]. Due to the heterogeneous delay of event propagation, the observed order of events may not be the same as the order in which the events were generated. Moreover, existing event stream processing schemes mainly focus on computing aggregates and statistics over the event streams, such as the sum and the median. Sampling and sketching techniques are adopted to cope with the huge amount and fast arrival of events [11], [35]. Our work differs from existing event processing schemes in both aspects discussed above. First, in our motivating scenarios, typically as monitoring an asynchronous distributed system, we do not assume the availability of synchronized clocks among the processes of a distributed system with message passing between each other. We intend to explicitly cope with the uncertainty caused by the asynchrony. Specifically, only the happen-before relation

among the events is available, which is a partial order relation. Thus, the events of different processes in the local windows cannot be sorted as a total order. By maintaining the Cartesian product of local windows, we can get the lattice of snapshots within the n-dimensional sliding window, which captures all possible interleavings of the events in the local windows. Based on the windowed lattice, we can further evaluate the predicates over all these possible interleavings. On the contrary, the union of local windows [10] can only get one possible interleaving of the events, which is a special case of ours. Second, as for the events of concern, we mainly focus on (usually global) patterns of events, such as the concurrency properties, the regular expression properties, and the temporal logic properties [6], [7], [8]. Also note that our scheme and the existing schemes are complementary and can be integrated to better process asynchronous event streams.

8 Conclusion and Future Work

In this work, we study the processing of asynchronous event streams within sliding windows. We first characterize the lattice structure *Lat-Win* of event stream snapshots within the sliding windows. Then we characterize how *Lat-Win* evolves when the n-dimensional sliding window slides over the asynchronous event streams, based on which we propose an online algorithm to maintain *Lat-Win*. The *Lat-Win* maintenance algorithm is implemented and evaluated over MIPA.

In our future work, we will study how to maintain time-based sliding windows, when the distributed event sources form a partially synchronous system with a metric time model. We will also study the approximate/probabilistic detection of specified predicates over asynchronous event streams.

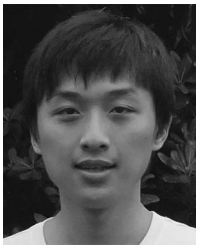
Acknowledgments

This work is supported by the National 973 Program of China (2009CB320702) and the National Natural Science Foundation of China (Nos. 61272047, 91318301, 61321491). Y. Huang is the corresponding author.

References

[1] W. Wolf, ''Cyber-Physical Systems,'' *IEEE Comput.*, vol. 42, no. 3, pp. 88-89, Mar. 2009.
[2] P.S. Duggirala, T.T. Johnson, A. Zimmerman, and S. Mitra, ''Static and Dynamic Analysis of Timed Distributed Traces,'' in *Proc. 33rd IEEE RTSS*, Dec. 2012, pp. 173-182.
[3] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen, ''Performance Debugging for Distributed Systems of Black Boxes,'' in *Proc. ACM SOSP*, 2003, pp. 74-89.
[4] J. Lu, X. Ma, X. Tao, C. Cao, Y. Huang, and P. Yu, ''On Environment-Driven Software Model for Internetware,'' *Sci. China Ser. F, Inf. Sci.*, vol. 51, no. 6, pp. 683-721, June 2008.
[5] F. Yang, J. Lu, and H. Mei, ''Technical Framework for Internetware: An Architecture Centric Approach,'' *Sci. China Ser. F, Inf. Sci.*, vol. 51, no. 6, pp. 610-622, June 2008.
[6] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, ''Runtime Detection of the Concurrency Property in Asynchronous Pervasive Computing Environments,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 744-750, Apr. 2012.
[7] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu, ''Formal Specification and Runtime Detection of Dynamic Properties in Asynchronous Pervasive Computing Environments,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1546-1555, Aug. 2013.
[8] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu, ''Formal Specification and Runtime Detection of Temporal Properties for Asynchronous Context,'' in *Proc. IEEE Int'l Conf. PerCom*, 2012, pp. 30-38.
[9] Y. Zhu and D. Shasha, ''Statstream: Statistical Monitoring of Thousands of Data Streams in Real Time,'' in *Proc. 28th Int'l Conf. VLDB*, 2002, pp. 358-369.
[10] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster, ''Prediction-Based Geometric Monitoring Over Distributed Data Streams,'' in *Proc. ACM SIGMOD Int'l Conf. Manag. Data, ser. SIGMOD'12*, 2012, pp. 265-276.
[11] S. Tirthapura, B. Xu, and C. Busch, ''Sketching Asynchronous Streams over a Sliding Window,'' in *Proc. 25th Annu. ACM Symp. Principles Distrib. Comput.*, 2006, pp. 82-91.
[12] A.D. Kshemkalyani and J. Cao, ''Predicate Detection in Asynchronous Pervasive Environments,'' *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1823-1826, Sept. 2013.
[13] A. Kshemkalyani, ''A Fine-Grained Modality Classification for Global Predicates,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 8, pp. 807-816, Aug. 2003.
[14] Y. Huang, J. Cao, B. Jin, X. Tao, J. Lu, and Y. Feng, ''Flexible Cache Consistency Maintenance over Wireless Ad Hoc Networks,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1150-1161, Aug. 2010.
[15] W. Wu, J. Cao, and X. Fan, ''Design and Performance Evaluation of Overhearing-Aided Data Caching in Wireless Ad Hoc Networks,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 3, pp. 450-463, Mar. 2013.
[16] O. Babaoğlu and K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*. New York, NY, USA: ACM Press, 1993.
[17] R. Schwarz and F. Mattern, ''Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,'' *Distrib. Comput.*, vol. 7, no. 3, pp. 149-174, Mar. 1994.
[18] L. Lamport, ''Time, Clocks, the Ordering of Events in a Distributed System,'' *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
[19] F. Mattern, ''Virtual Time and Global States of Distributed Systems,'' in *Int'l Workshop Parallel Distrib. Algorithms*, 1989, pp. 215-226.
[20] O. Babaoğlu, E. Fromentin, and M. Raynal, ''A Unified Framework for the Specification and Run-Time Detection of Dynamic Properties in Distributed Computations,'' *J. Syst. Softw.*, vol. 33, no. 3, pp. 287-298, June 1996.
[21] C. Jard, G. Jourdan, T. Jeron, and J. Rampon, ''A General Approach to Trace-Checking in Distributed Computing Systems,'' in *Int'l Conf. Distrib. Comput. Syst.*, 1994, pp. 396-403.
[22] V. Braverman, R. Ostrovsky, and C. Zaniolo, ''Optimal Sampling From Sliding Windows,'' in *Proc. ACM SIGMOD-SIGACT-SIGART Symp. PODS*, 2009, pp. 147-156.
[23] M. Datar, A. Gionis, P. Indyk, and R. Motwani, ''Maintaining Stream Statistics Over Sliding Windows: (Extended abstract),'' in *Proc. ACM-SIAM Symp. Discr. Algorithms*, 2002, pp. 635-644.
[24] MIPA—Middleware Infrastructure for Predicate Detection in Asynchronous Environments. [Online]. Available: http://mipa.googlecode.com
[25] V. Garg and B. Waldecker, ''Detection of Strong Unstable Predicates in Distributed Programs,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1323-1333, Dec. 1996.
[26] V.K. Garg and B. Waldecker, ''Detection of Weak Unstable Predicates in Distributed Programs,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 3, pp. 299-307, Mar. 1994.
[27] F. Niederman, R. Mathieu, R. Morley, and I. Kwon, ''Examining RFID Applications in Supply Chain Management,'' *Commun. ACM*, vol. 50, no. 7, pp. 92-101, July 2007.
[28] V.K. Garg, N. Mittal, and A. Sen, ''Applications of Lattice Theory to Distributed Computing,'' *ACM SIGACT Notes*, vol. 34, no. 3, pp. 40-61, Sept. 2003.
[29] G.A. Grätzer, *General Lattice Theory* 2nd ed., Basel, Switzerland: Birkhäuser, 2003.
[30] R. Cooper and K. Marzullo, ''Consistent Detection of Global Predicates,'' in *Proc. ACM/ONR PADD*, 1991, pp. 167-174.
[31] A. Sen and V. Garg, ''Formal Verification of Simulation Traces Using Computation Slicing,'' *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 511-527, Apr. 2007.
[32] N. Mittal and V.K. Garg, ''Techniques and Applications of Computation Slicing,'' *Distrib. Comput.*, vol. 17, no. 3, pp. 251-277, Mar. 2005.

[33] L.-P. Chen, D.-J. Sun, and W. Chu, ''Efficient Online Algorithm for Identifying Useless States in Distributed Systems,'' *Distrib. Comput.*, vol. 23, no. 5/6, pp. 359-372, Apr. 2011.
[34] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, ''Models and Issues in Data Stream Systems,'' in *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2002, pp. 1-16.
[35] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang, ''Continuous Sampling from Distributed Streams,'' *J. ACM*, vol. 59, no. 2, pp. 10:1-10:25, Apr. 2012.

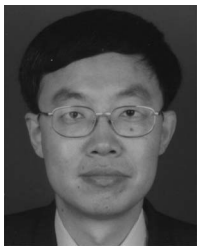


Yiling Yang received the BSc degree in computer science from Nanjing University, China, in 2009, where he is now working toward the PhD degree in computer science. His research interests include software engineering and methodology, theory of distributed computing, middleware technologies and pervasive computing. He is a Student Member of the IEEE.

Yu Huang received the BSc and PhD degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2002 and 2007, respectively. From 2003 to 2007, he studied in the Institute of Software, Chinese Academy of Sciences, as a co-educated PhD student. He also studied in the Department of Computing, Hong Kong Polytechnic University, as an exchange student from Sept. 2005 to Sept. 2006. He is currently an associate professor in the Department of Computer Science and Technology, Nanjing University, China. His research interests include distributed computing theory, formal specification and verification, and pervasive context-aware computing. He is a member of China Computer Federation. He is a member of the IEEE.

Jiannong Cao received the BSc degree in computer science from Nanjing University, China, in 1982, and the MSc and PhD degrees in computer science from Washington State University, Pullman, Washington, USA, in 1986 and 1990, respectively. He is currently a chair professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. Before joining Hong Kong Polytechnic University, he was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile and wireless computing, fault tolerance, and distributed software architecture. He has published over 300 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing test-bed, protocols, middleware and applications. Dr. Cao is a senior member of China Computer Federation, a senior member of the IEEE, including Computer Society and the IEEE Communication Society, and a member of ACM. He is also a member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/program committee member for many international conferences. He is a Senior Member of the IEEE.

Xiaoxing Ma received the BSc and PhD degrees from Nanjing University, China, both in computer science. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His research interests include software methodology and software adaptation. He is a member of the IEEE.



Jian Lu received the BSc, MSc, and PhD degrees in computer science from Nanjing University, China. He is currently a Professor in the Department of Computer Science and Technology and the Director of the State Key Laboratory for Novel Software Technology at Nanjing University. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems.